Certified JSON Derivers with ELPI

WILL THOMAS, University of Kansas, USA

Verified Rocq components often communicate through ordinary formats after extraction. When two verified components exchange a value through ad hoc JSON encoders and decoders, the components can be correct while the pipeline fails because the boundary code changes, drops, or misreads the value. This paper presents `rocq-json`, an ELPI-based deriver that generates JSON encoders, decoders, and Rocq-checked round-trip proofs for ordinary inductive and record types. We present the concrete contract the tool provides and the performance profile of the generated serializers.

1 MOTIVATION

Consider a small verified pipeline with two extracted Rocq components [2]. The first component produces a structured value: for example a counterexample trace, an event log summary, or a certificate for a second checker. The second component consumes that value and has its own Rocq proof about what it does with well-formed inputs. If the two components are connected by hand-written JSON serialization code, the verified parts are no longer the whole story. An encoder can omit a constructor argument, choose a different field name, or encode a constructor differently from what the decoder expects. The result is a pipeline that can fail, or worse consume a different value, even though both Rocq components are internally correct.

`rocq-json` provides exactly the missing boundary contract for this producer-to-consumer case. Instead of separately maintaining an ad hoc encoder and decoder, the user derives them from the Rocq declaration together with a checked `Jsonifiable` instance:

```
Class Jsonifiable (A : Type) := {
  to_JSON   : A → JSON;
  from_JSON : JSON → Result A string;
  canonical_jsonification : forall a, from_JSON (to_JSON a) = res a
}.
```

The theorem says that any Rocq value sent through the generated encoder is recovered by the generated decoder. It rules out the class of bugs where the tool boundary itself loses or changes the value.

2 TOOL

The user-facing interface is intentionally small: define a datatype and invoke the existing `coq-elpi` derive framework. For example, a component that emits roles for a downstream authorization checker can make the JSON boundary checked with one command:

```
Inductive UserRole : Type :=
| Admin | Guest | Moderator (level : nat)
| StandardUser (id : nat) (username : string).

Elpi derive.jsonifiable UserRole.

Example moderator_json :
  to_JSON (Moderator 2) =
    JSON_Object [("Moderator", JSON_Object [("level", JSON_Nat 2)])] := eq_refl.
```

A successful command registers an instance containing an encoder, decoder, and proof. A failed command leaves the environment unchanged and reports the first recognized unsupported shape.

Author's address: Will Thomas, University of Kansas, Lawrence, USA, 30wthomas@ku.edu.

In the example, nullary roles become JSON strings, and field-bearing roles become singleton objects tagged by constructor name with a nested object for their fields.

Records use projection names as object keys. Parameterized datatypes receive typeclass instance arguments for their parameters, so generated serializers compose with primitive, container, and user-defined serializers.

The implementation uses `coq-elpi` [1] to inspect declarations and construct Rocq terms for the encoder, decoder, proof, and final typeclass instance. Rocq checks those generated terms before they enter the development. The metaprogram can fail or choose an inconvenient representation, but it cannot by itself establish a false round-trip theorem.

3 EVALUATION

The standard library `Stdlib` scan is useful mainly as context: after excluding declarations that are not closed inputs and deliberate non-targets such as `Prop`, functions, sorts, and coinductive types, the run derives 81 of 153 target-fragment inputs (53%). The remaining gaps are dependent fields, indexed families, and proof-bearing computational wrappers. The more interesting question is whether successful derivations are quick and whether the extracted code is plausible.

Table 1. Derivation and extracted-code speed results.

Metric	Value
Successful derivations	81
Full benchmark	5.26 s
Derivation sum	4.34 s
Mean	0.054 s
Median	0.033 s
90th percentile	0.101 s
Max	0.500 s

Shape	Runtime Derived (s)	Handwritten (s)	Change
enum256	43.917	43.052	+2.0%
point2d	1.066	1.170	-8.9%
userrole	1.790	2.648	-32.4%
serverconfig	0.635	1.525	-58.4%
instruction	1.733	2.666	-35.0%
bintree	3.321	5.301	-37.3%

Table 1 shows that derivation is fast enough for interactive use; the slowest case is `byte`, a 256-constructor enumeration. It also compares generated definitions with handwritten Rocq serializers extracted through the same pipeline. The generated code is within 2.0% on the large enumeration and faster on the other measured shapes, including the `UserRole` example. The point is not optimal JSON performance; it is that a checked deriver removes an ad hoc serialization boundary without imposing an obvious extraction-time penalty.

REFERENCES

[1] Enrico Tassi. 2025. Elpi: rule-based meta-language for Rocq. In *CoqPL 2025 - The Eleventh International Workshop on Coq for Programming Languages*. Denver, United States. https://inria.hal.science/hal-04990628

[2] The Rocq Development Team. 2025. The Rocq Prover. Zenodo, Version 9.0.0, 10.5281/zenodo.11551307. https://doi.org/10.5281/zenodo.15149629